Государственное образовательное учреждение высшего образования
Санкт Петербургский государственный технологический институт (технический университет)

Кафедра систем автоматизированного проектирования и управления

Лингвистическое и программное обеспечение автоматизированных систем

Методические указания к выполнению контрольных работ для студентов заочной формы обучения

Санкт-Петербург
2016

Введение

Основная проблема метода функциональной декомпозиции заключается в том, что он не позволяет каким-либо образом подготовить текст программы к последующим изменениям, появляющимся по мере её постепенной эволюции. Чаще всего подобные изменения связаны с необходимостью учесть новый вариант поведения программы в уже существующей системе. Если всю логику реализации поэтапной схемы, которая получается при функциональной декомпозиции, поместить в одну большую функцию или модуль, то каждое изменение любого из этапов потребует внесения изменений в эту функцию или модуль.

Множество ошибок появляется при внесении в текст программы изменений. Всякий раз, когда в текст программы необходимо внести изменения, возникает опасение, что изменение текста программы в одном месте может привести к сбою в другом. Возникает множество вопросов – должен ли программист об этом думать, должен ли исследовать взаимодействие модулей и функций и т.д.

Важно понимать: ничто не может предотвратить наступление перемен. Но это не означает, что к их приходу нельзя подготовиться.

Любой опрос среди разработчиков программного обеспечения по качеству предоставляемых им заказчиком требований к создаваемому программному обеспечению едва ли даст большое разнообразите ответов. Скорее всего, они будут следующими: неполные, по большей части ошибочны, противоречивы, не описывают поставленную задачу подробно.

Результат: требования к программному обеспечению всегда меняются.

Причины:

- Взгляды пользователей на их нужды меняются в ходе бесед с разработчиками, в ходе использования появляющейся системы
- Представление разработчиков о предметной области меняется по мере создания ими программного обеспечения.
- Появляются новые вычислительные среды, предназначенные для разработки ПО.

За исключением самых простых случаев, требования изменяются всегда и везде, вне зависимости от того, насколько хорошо был сделан первичный анализ. Вместо того, чтобы жаловаться на изменение исходно представленных требований, необходимо так модифицировать процесс разработки, чтобы обеспечить эффективную обработку любых возникающих изменений.

Почему же?

Существенная черта промышленной программы – уровень сложности: один разработчик практически не в состоянии охватить все аспекты такой системы. Эта сложность неизбежна: с ней можно справиться, но избавиться от неё нельзя.

Сложность вызывается четырьмя основными причинами:

- сложностью реальной предметной области, из которой исходит заказ на разработку
- трудностью управления процессом разработки
- необходимостью обеспечить достаточную гибкость программы
- неудовлетворительными способами описания поведения больших дискретных систем.

При этом основная задача разработчиков состоит в создании иллюзии простоты, в защите пользователей от сложности описываемого предмета или процесса.

Структура сложных систем

Примеры: структура компьютера, структура растения, структура человека.

Иерархическая структура: каждая часть состоит из взаимодействующих меньших частей, из которых собирается целое, как верхний уровень с возникающим поведением.

Система – не только модули, её образующие, но и связи между ними. И топология этих связей. Дело не только в том, что система иерархична, но в том, что уровни этой

иерархии представляют различные уровни абстракции, причём один надстроен над другим и каждый может быть рассмотрен (понят) отдельно.

Только благодаря совместному действию большого числа посредников образуется более высокий уровень функционирования системы. Наука о сложности называет это возникающим поведением. Поведение целого сложнее, чем поведение суммы его составляющих.

Для каждого уровня абстракции чётко разграничено «внутреннее» и «внешнее». Например, можно установить, что части листа совместно обеспечивают функционирование листа в целом и очень слабо взаимодействуют или вообще прямо не взаимодействуют с элементами корней. Проще говоря, существует чёткое разделение функций различных уровней абстракций.

Многие системы (животные, растения) имеют сходства в построении систем, которые позволяют и классифицировать, создавая другой вид иерархии – классификацию.

Пять признаков сложной системы

1. Сложные системы часто являются иерархическими и состоят из взаимозависимых подсистем, которые в свою очередь также могут быть разделены на подсистемы и т.д., вплоть до самого низкого уровня.
2. Архитектура сложных систем складывается и из компонентов, и из иерархических отношений этих компонентов. Особенности системы обусловлены отношениями между её частями, а не частями, как таковыми. Выбор, какие компоненты в данной системе считаются элементарными, относительно произволен и в большей степени оставляется на усмотрение исследователя.
3. Внутрикомпонентная связь обычно сильнее, чем связь между компонентами. Это обстоятельство позволяет отделять «высокочастотные» взаимодействия внутри компонентов от «низкочастотной» динамики взаимодействия между компонентами.
4. Иерархические подсистемы обычно состоят из немногих типов подсистем, по-разному скомбинированных и организованных.
5. Любая работающая система является результатом развития работавшей более простой системы. Сложная система, спроектированная «с нуля», никогда не заработает. Следует начинать с работающей простой системы.

Различие алгоритмической и объектно-ориентированной декомпозиции.

Задача разделяется не на шаги, которые должны быть выполнены последовательно (алгоритм), а на группу взаимодействующих объектов, каждый из которых наделён собственной зоной ответственности. И эти объекты во взаимодействии решают общую задачу.

Объектно-ориентированный анализ – это методология, при которой требования к системе воспринимаются с точки зрения классов и объектов, выявленных в предметной области.

Объектно-ориентированное проектирование – это методология проектирования, соединяющая в себе процесс объектной декомпозиции и приёмы представления логической и физической, а также статической и динамической моделей проектируемой системы.

Объектно-ориентированное программирование – это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.

Основные элементы объектной модели

Объект – физическая или умозрительная сущность, облегчающая понимание реального мира и имеющее чётко определённое функциональное назначение в данной предметной области. Объект моделирует часть окружающей действительности и таким образом существует во времени и пространстве.

Преимущество использования объектов заключается в том, что на них возлагается ответственность за их собственное поведение. Объекты изначально относятся к определённому типу, известному им. Данные объекта позволяют ему определять своё состояние, а программный код (методы) обеспечивает его корректное функционирование (т.е. выполнение тех действий, для которых он, собственно, предназначен).

На концептуальном уровне объекты выступают как совокупность обязательств.

На уровне определения спецификаций объекты понимаются как набор методов, которые могут вызываться другими объектами или этим же объектом.

На уровне реализации объекты рассматриваются как совокупность программного кода и данных.

Класс – есть описание структуры объекта и его интерфейса. Представляет контракт между объектом и всеми его клиентами.

Абстракция выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов и таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя.

Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности поведения от несущественных.

Инкапсуляция – это процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение; инкапсуляция служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации.

Наследование – это такое отношение между классами, когда один класс повторяет структуру и поведение другого класса (одиночное наследование) или других (множественное наследование) классов. Наследование есть механизм разделения и повторного использования описания структуры объекта (данные класса) и его поведения (методы класса).

Класс, поведение и структура которого наследуется называется базовым (родительским) классом, а класс, который наследует – производным.

В производном классе структура и поведение базового класса дополняются и переопределяются. Производный класс является уточнением базового класса.

Наследование есть классификация. Классификация позволяет повторно использовать единожды осмысленные и реализованные свойства класса.

Полиморфизм

Возможность одним и тем же способом взаимодействовать с объектами различных типов. При этом достигаемый результат будет зависеть от типа объекта, к которому производится обращение.

В данном учебном пособии представлены задания на выполнение контрольных работ по изучению основных принципов объектно-ориентированного программирования и созданию простых программ для их иллюстрации в инструментальной среде разработки приложений на языке С++.

Задание на выполнение контрольных работ

В учебном пособии составлены 3 контрольные работы. Студенту необходимо представить отчёт о выполненных контрольных работах в распечатанном виде и в электронном виде на любом носителе информации.

Отчёт должен включать: титульный лист, условие задачи, алгоритм решения (при необходимости), программу и результаты. Во время защиты контрольных работ студент должен подтвердить работоспособность программной реализации заданий. На титульном листе отчёта о выполнении контрольных работ необходимо указать фамилию, имя и отчество студента, номер учебной группы, номер контрольной работы, номер варианта.

Номер варианта соответствует номеру первой буквы фамилии студента согласно таблице 1.

Таблица 1 — Распределение вариантов заданий

Первая буква фамилии студента	**Номер варианта**	**Первая буква фамилии студента**	**Номер варианта**
А, Б	1	Р, С	9
В, Г	2	Т, У	10
Д, Е, Ё	3	Ф, Х	11
Ж, З	4	Ц, Ч	12
И, Й	5	Ш, Щ	13
К, Л	6	Ы, Э	14
М, Н	7	Ю, Я	15
О, П	8	Пример решения	16

Приступая к выполнению контрольных работ, рекомендуется ознакомиться со следующими методическими материалами:

1) Тенишев, Д. Ш. Лингвистическое и программное обеспечение автоматизированных систем : учеб. пособие для вузов / Д. Ш. Тенишев ; под ред. Т. Б. Чистяковой. – СПб. : ЦОП «Профессия», 2010. – 403 с.

2) Волосатова Т.М., Родионов С.В. Автоматизация проектирования лексических анализаторов. – М.: Издательство МГТУ им. Баумана, 2005.

3) Опалева Э.А., Самойленко В.П. Языки программирования и методы трансляции. – СПб.: БХВ-Петербург, 2005.

Контрольная работа №1. Теоретические вопросы по курсу лингвистическое и программное обеспечение автоматизированных систем

Вариант 1

1. Перечисления (enum) в C++
2. Понятие объекта в объектно-ориентированном программировании. Отношения между объектами.

Вариант 2

1. Классы в языке C++. Объявление и определение. Ключевое слово this.
2. Структуры в языке C++. Определение и объявление. Выравнивание данных в структурах.

Вариант 3

1. Функции в C++: объявление, передача параметров, возвращаемое значение, аргументы по умолчанию, неуказанное число аргументов, указатель на функцию.
2. Множественное наследование в языке C++. Абстрактные классы.

Вариант 4

1. Функциональная декомпозиция. Примеры применения. Достоинства и недостатки. Восходящее и нисходящее проектирование.
2. Полиморфизм в объектно-ориентированном программировании. Статический полиморфизм в C++. Примеры применения.

Вариант 5
1. Фундаментальные типы данных C++.
2. Указатели и массивы в C++. Инициализаторы массивов. Доступ к элементам массивов. Многомерные массивы.

Вариант 6
1. Объектно-ориентированное программирование. Полиморфизм. Динамический полиморфизм в C++. Примеры применения.
2. Объектная модель. Объектно-ориентированный анализ. Объектно-ориентированное проектирование. Основные инструменты объектной модели.

Вариант 7
1. Инкапсуляция в объектно-ориентированном программировании. Примеры применения.
2. Классы в языке C++. Отношения между классами.

Вариант 8
1. Выражения и операторы. Приоритет и ассоциативность. Действие и результат операторов. Постфиксные и префиксные операторы. Арифметические преобразования в выражениях.
2. Объектно-ориентированное программирование. Понятие объекта и класса в объектно-ориентированном программировании. Абстрагирование.

Вариант 9
1. Модульное тестирование программного обеспечения. Механизмы реализации. Примеры.
2. Область видимости и время жизни объектов в C++. Объявление и определение объектов. Создание и удаление объектов.

Вариант 10
1. Разделение программы на языке C++ на файлы с исходным кодом. Включаемые заголовки. Директива include. Использование макросов условной компиляции. Причины разделения на модули. Модульные интерфейсы.
2. Ссылки в языке C++. Случаи использования.

Вариант 11
1. Константы. Константные методы. Обоснование использования. Случаи использования. Синтаксис. Указатели на константы и константные указатели.
2. Пространства имён (namespace) в языке C++. Анонимные пространства имён. Директива using. Объявление using.

Вариант 12
1. Тенденции развития проектирования и языков программирования (математические, структурные, модульные, объектно-ориентированные, обобщенные, событийно-управляемые).
2. Использование утверждений в C++ (assert).

Вариант 13
1. Классы в языке C++. Создание и удаление объектов. Конструкторы и деструкторы. Статические члены класса.
2. Одиночное наследование в языке C++. Абстрактные классы.

Вариант 14
1. Объявления typedef в C++
2. Строки в стиле языка программирования C. Основные алгоритмы стандартной библиотеки для работы со строками в стиле языка C.

Вариант 15
1. Одиночное наследование в языке C++. Перегрузка методов, переопределение методов, сокрытие методов.
2. Принципы обработки ошибок в программе на C++. Возвращаемые значения, глобальные флаги состояний, исключения.

Пример выполнения контрольной работы № 1. Вариант № 16

1. Динамическое распределение памяти

Время жизни именованного объекта определяется его областью видимости. С другой стороны часто возникает необходимость в создании объектов, которые существуют вне зависимости от области видимости, в которой они созданы.

Одним из решений в данном случае может являться использование глобальных объектов, время жизни которых равно времени жизни программы. Однако глобальные объекты не решают других задач:

1. Возможности создавать объект только тогда, когда он необходим (например, при возвращении объекта как результата работы функции)
2. Возможности создавать массив объектов заранее неизвестного размера, т.е. в том случае, когда размер массива становится известен в ходе выполнения программы.

Для решения таких задач лучше всего подходит механизм динамического создания объектов из заранее выделенной области свободной памяти. Такая память называется «кучей» (heap) и выделяется операционной системой и системными библиотеками при запуске программы. Это неименованная область памяти, которая при запуске принадлежит операционной системе (или программе) и не принадлежит никакому конкретному объекту в программе.

Для работы с динамической памятью («кучей») используются операторы `new` и `delete`.

Оператор `new` позволяет динамически выделить часть памяти из «кучи» и получить указатель на выделенный участок памяти.

У оператора `new` есть две формы: `new` и `new[]`.

Первая форма позволяет распределить память для одного объекта заданного типа и имеет следующий синтаксис:

```
T* new T
```

Оператор new выделяет память для одного объекта типа `T` и возвращает указатель на выделенный участок памяти. Указатель имеет тип `T*`.

Пример:

```
Circle *p;
p = new Circle;
```

Вторая форма оператора `new` позволяет выделить память для распределения массива объектов:

```
T* new T[size]
```

Оператор `new[]` выделяет непрерывный участок памяти для `size` объектов типа `T` и возвращает указатель на выделенный участок памяти.

Пример:

```
Circle *array;
array = new Circle[nCircles];
```

Обе формы оператора new обеспечивают вызов конструкторов создаваемых объектов пользовательских типов.

Для объектов, созданных в динамической памяти неявной инициализации не производится, то есть в случае, если явно не указано начальное значение для создаваемого объекта, то начальное значение будет не определено («мусор»).

Для освобождения ранее выделенной оператором new памяти используется оператор delete. Он возвращает ранее выделенную память обратно в «кучу» с тем, чтобы её можно было использовать для создания других объектов.

У оператора delete также существует две формы: delete и delete[].

Первая форма используется для освобождения памяти, распределённой оператором new:

Пример:

```
Circle *p;
p = new Circle;
…
delete p;
```

Вторая форма используется для освобождения памяти, распределённой оператором new[]:

Пример:

```
Circle *array;
array = new Circle[nCircles];
…
delete[] p;
```

Обе формы оператора delete обеспечивают вызов деструкторов создаваемых объектов пользовательских типов.

Память, выделенная оператором new должна освобождаться только оператором delete, а память, выделенная оператором new[] должна освобождаться только оператором delete[]. Нарушение этого правила является ошибкой и ведёт к непредсказуемому поведению программы.

В случае, если в операторе new запросить выделить больше памяти, чем доступно программе, возможны несколько вариантов поведения:

- Оператор new вернёт нулевой указатель, показывая тем самым, что выделение памяти прошло неудачно и память не распределена. Это поведение по умолчанию в программе, скомпилированной в Visual Studio.NET.
- Оператор new сгенерирует исключение bad_alloc, указывая на аварийную ситуацию. Это поведение по умолчанию по стандарту языка. В Visual Studio такого поведения программы можно добиться, подключив стандартный заголовочный файл <new>.
- Будет вызвана заданная пользователем функция, если она была указана вызовом set_new_handler().

В случае, если в операторе new запросить массив нулевого размера (с нулевым числом элементов), то оператор new вернёт уникальный адрес памяти, но использовать его для записи или чтения будет нельзя, поскольку реально зарезервировано ноль байт.

Повторное применение оператора delete к одному и тому же указателю является ошибкой и приведёт к непредсказуемому поведению программы.

Применение оператора delete к нулевому указателю (указателю, значением которого является ноль), не приводит к выполнению каких-либо действий. То есть это «пустая» операция, которая не приведёт к ошибке. (Указанное правило верно для встроенных типов и пользовательских типов, у которых не перегружены операторы new и delete).

В старых программах на C для выделения памяти из кучи используются функции `malloc` и `free`. Их не следует использовать в программах на C++, поскольку они не обеспечивают безопасности в отношении типов, не вызывают деструкторов и имеют ряд других недостатков.

Смешивать управление памяти по `new`/`delete` и `malloc`/`free` недопустимо. То есть, память, выделенную по одному из этих механизмов нельзя освобождать другим. Это является ошибкой и ведёт к непредсказуемому поведению программы.

2. Шаблоны структур

Шаблоны структур, так же, как и шаблоны функций, описывают правило генерации кода для пользователей типов. В случае структур тип, используемый в параметре шаблона должен задаваться пользователем явно при инстанцировании.

Например:

```
template <typename T1, typename T2>
struct pair {
  T1 first;
  T2 second;
};
```

позволит использовать его следующим образом:

```
pair<int,int> cursor_position;
```

что эквивалентно:

```
struct {
  int first;
  int second;
} cursor_position;
```

К сожалению, понятностью кода в данном случае придётся пожертвовать. Но это позволяет удобные расширения при описании данных. Имя один шаблон, описывающий пару элементов:

```
template <typename T1, typename T2>
struct pair {
  T1 first;
  T2 second;
};
```

Можно в любой момент получить новый тип, не описывая его явно.

```
typedef pair<bool,pair<float,float> > CrossPoint;
```

В данном случае описана пара, состоящая из переменной логического типа и пары, которая в свою очередь состоит из двух переменных типа float.

```
CrossPoint FindIntersection() {
  CrossPoint pt;
  pt.first = true;
  pt.second.first  = 0.0f;
  pt.second.second = 1.0f;
  return pt;
}

int main(){
  CrossPoint pt;
  pt = FindIntersection();
}
```

Такой подход бывает удобен, когда ввести составной тип в очень локальном участке программы, где можно поступиться понятностью кода в пользу его компактности. Либо при разработке алгоритмов для работы с абстрактными типами данных, когда на стадии разработки алгоритма контекст его использования неизвестен.

Контрольная работа №2. Наследование и полиморфизм

На основе базового класса «книга» (поля: название, год издания) создать заданный производный класс, который, помимо перечисленных, содержит данные об авторе и количестве страниц. Описать виртуальную функцию вывода на экран информации о книге.

№	Производный класс
1	научно-популярная литература
2	программная документация
3	учебная литература
4	программная документация
5	журнал
6	программная документация
7	журнал
8	учебная литература

№	Производный класс
9	журнал
10	художественная литература
11	учебная литература
12	художественная литература
13	научно-популярная литература
14	художественная литература
15	научно-популярная литература
16	руководство программисту

Пример выполнения программы. Контрольная работа № 2, вариант № 16.

Создание производного класса «руководство программисту». Код программы:

```
#include <iostream>
using namespace std;

//базовый класс "книга"
class book {
  char* title; //название книги
  int year;    //год издания
public:
  book(char*, int); //конструктор
  virtual void display(); //виртуальная функция печати
};

//производный класс "руководство программиста"
class programmers_guide : public book {
  int page_count; //количество страниц
  char* author; //автор книги
public:
  programmers_guide(char*, int, int, char*); //конструктор
  void display(); //вывод на экран
};

//конструктор базового класса
book::book(char* book_title, int book_year)
  :title(book_title), year(book_year)
{ }

//конструктор производного класса
programmers_guide::programmers_guide(char* book_title,
  int book_year, int page_count, char* book_author)
  //вызов конструктора базового класса
```

```
  :book(book_title, book_year),
  page_count(page_count), author(book_author)
{ }

//функция печати для базового класса
void book::display() {
  cout << title << ", " << year << endl;
}

//функция печати для производного класса
void programmers_guide::display() {
  book::display(); //вызов функции-предка
  cout << "Количество страниц: " << page_count << "\nАвтор - "
       << author << endl;
}

void main() {
  setlocale(0, "");
  book b("Oxford Advanced Learner's Dictionary", 1994),
       *pointer;
  programmers_guide pg("Object Windows for C++", 1993, 300,
                       "Dennis MacAlistair Ritchie");
  pointer = &b;  //указатель на объект базового класса
  pointer->display(); //функция базового класса
  pointer = &pg; //указатель на объект производного класса
  pointer->display(); //функция производного класса
  system("PAUSE");
}
```

Контрольная работа №3. Абстрактные классы и виртуальные функции

Создать абстрактный базовый класс «число» с виртуальной функцией вычисления модуля. Определить заданный производный класс со своей функцией модуля. Для проверки определить массив указателей на абстрактный класс, элементами которого являются указатели на объекты производных классов.

№	Производный класс
1	рациональное число
2	целое число
3	натуральное число
4	вещественное число
5	натуральное число
6	рациональное число
7	вещественное число
8	рациональное число

№	Производный класс
9	вещественное число
10	целое число
11	натуральное число
12	целое число
13	рациональное число
14	натуральное число
15	целое число
16	комплексное число

Пример выполнения программы. Контрольная работа № 3, вариант № 16.

Создание производного класса «комплексное число». Код программы:

```
#include <iostream>
using namespace std;

//базовый абстрактный класс "число"
class number {
protected:
  float real;
public:
  number(float re) { //конструктор
          real = re;
  }
  //чистая виртуальная функция вычисления модуля
  virtual float module() = 0;
};

//производный класс "действительное число"
class real_numb :public number {
public:
  real_numb(float re = 0.0) :number(re) { }
  float module() { //вычисление модуля
    return real >= 0 ? real : -real;
  }
};

//производный класс "комплексное число"
class complex_numb :public number {
  float image; //мнимая часть
public:
  complex_numb(float re = 0.0, float im = 0.0) :number(re) {
    image = im;
  }
  float module() { //вычисление модуля
    return sqrt(pow(real, 2) + pow(image, 2));
  }
};

void main() {
  setlocale(0, "");
  //массив указателей на абстрактный базовый класс
  number *NumberArray[5];
  //элементы массива - объекты производных классов
  NumberArray[0] = new real_numb(1.2);
  NumberArray[1] = new complex_numb(-3.0, 4.0);
  NumberArray[2] = new real_numb;
  NumberArray[3] = new complex_numb;
  NumberArray[4] = new complex_numb(1.0);
  //вызов функции модуля для каждого элемента массива
  for (int i = 0; i < 5; i++)
    cout << "Модуль числа N" << i + 1 << " = "
         << NumberArray[i]->module() << endl;
  system("PAUSE");
}
```